A Surprisingly Simple Lua Compiler

Hugo Musso Gualandi
hgualandi@inf.puc-rio.br
PUC-Rio

Roberto Ierusalimschy
roberto@inf.puc-rio.br
PUC-Rio

ABSTRACT

Dynamically-typed programming languages are often implemented using interpreters, which offer several advantages in terms of portability and flexibility of the implementation. However, as a language matures and its programs get bigger, programmers may seek compilers, to avoid the interpretation overhead.

In this study, we present LuaAOT, a simple ahead-of-time compiler for Lua. The compiler is derived from the Lua interpreter and it exemplifies an old idea of using partial evaluation to produce a compiler based on an existing interpreter. Our contribution is to apply this idea to a well-established programming language. We show that with a quite modest effort it is possible to implement an efficient compiler that covers the entirety of Lua, including coroutines and tail calls. The whole implementation required less than 500 lines of new code. For this effort, we reduced the running time of our benchmarks from 20% to 60%.

CCS CONCEPTS

• **Software and its engineering → Interpreters**; **Just-in-time compilers**.

KEYWORDS

dynamic languages, interpreters, partial evaluation, compilers, just-in-time compilers

1 INTRODUCTION

Dynamic programming languages are popular for many applications, including scripting. They are often implemented using an interpreter, which makes it easier to load code fragments at runtime and enables a fast test-change-recompile development loop. However, interpreters are often slower than a compiler, due to the interpretation overhead.

The conventional wisdom is that the most efficient implementations for dynamic languages are just-in-time compilers, which can take advantage of run-time information to perform speculative optimizations. Ahead-of-time compilers have a steeper hill to climb, because they must rely on clever static analysis or type inference to inform their optimizations. Nevertheless, in this paper we show that a very simple compiler, focused exclusively on reducing the interpretation overhead, can deliver respectable improvements for a modest implementation effort. We also argue that such a simple compiler can provide useful insight about the performance of the interpreter that it is based on.

Our motivation for this paper was our previous work on Lua compilers, in particular the Pallene language [10]. Pallene is superficially similar to a typed dialect of Lua, where the type information allows the compiler to perform significant optimizations to the code. Because the type information is not speculative, Pallene's compiler can work ahead of time and be much simpler than a JIT compiler. However, a relevant question is how much of this improvement is due to the types and how much of it is due to just using a compiler instead of an interpreter. To help answer this question, we developed LuaAOT, a simple ahead-of-time compiler for Lua which does not perform any type-based optimizations.

The inspiration for the architecture of LuaAOT is an old idea of producing a compiler from an existing interpreter by unrolling and specializing the core interpreter loop [7, 15]. Our contribution is to show that this idea can be successfully applied to an established language. Using less than 500 lines of new code, our compiler implements the entirety of Lua, including coroutines and tail calls.

One thing that contributes to the simplicity of LuaAOT is that we can delegate a significant part of the work to a C compiler. We can rely on the C compiler to perform several optimizations, including constant propagation and dead code elimination. This allows us to remove much of the interpreter overhead while still emitting straightforward code that is mostly copied from the existing interpreter.

Before going on, we should emphasize that LuaAOT catches the low-hanging-fruits of compiler optimizations for dynamic languages. Its performance is not competitive with a reasonable JIT compiler. Its selling point is that it achieves a decent performance boost for a surprisingly low cost.

The next section has a brief discussion about partial evaluation of virtual machines. Section 3 describes our take on that idea, LuaAOT. Next, we evaluate our artifact in Section 4. Finally, we discuss related work in Section 5 and draw some conclusions in Section 6.

2 VIRTUAL MACHINES AND PARTIAL EVALUATION

One of the most popular ways to implement an interpreter for a dynamically typed programming language is via a virtual machine. The virtual machine defines an intermediate language of portable instructions, also called *bytecodes*. This approach is illustrated in Figure 1, which shows a small Lua function and the corresponding portable instructions for the Lua virtual machine. (To ease the presentation, we represented these instructions using records. The actual Lua interpreter encodes the instruction components as bitfields of a 32-bit integer [11].)

Figure 2 shows a typical inner loop of a virtual machine. It executes the portable instructions, one by one, like a conventional CPU. The interpreter maintains a stack, which is where the local variables are stored. The program counter points to the current instruction and guides the control flow. Data operations, such as LOADI and ADD, manipulate the values in the stack. Control-flow operations, such as JUMP, modify the program counter. In this example, `DoAdd` is a macro that does the actual work, including checking the types of the arguments.

The virtual machine in our example is register-based, similarly to the Lua virtual machine [11]. The defining characteristic of a

```
function foo(a, b, c)
    local d = 17
    while true
        a = b + c
        b = b + d
    end
end

Instruction foo[] = {
    { LOADI, 3, 17 },
    { ADD, 0, 1, 2 },
    { ADD, 1, 1, 3 },
    { JUMP, 1 }
};
```

Figure 1: A Lua function and its bytecode.

```
void execute(Instruction prog[], Value stack[])
{
    int pc = 0;
    while (1) {
        Instruction instr = prog[pc++];
        switch (instr.tag) {
            case LOADI: {
                int dst = instr.arg1;
                int val = instr.arg2;
                stack[dst] = IntValue(val);
                break;
            }
            case ADD: {
                int dst  = instr.arg1;
                int src1 = instr.arg2;
                int src2 = instr.arg3;
                stack[dst] = DoAdd(stack[src1],
                                   stack[src2]);
                break;
            }
            case JUMP: {
                pc = instr.arg1;
                break;
            }
        }
    }
}
```

Figure 2: A virtual machine / interpreter.

register-based virtual machine is that the data-manipulation instructions can read from and write to any position in the stack. The other common way to design a virtual machine is in a stack-based discipline, where the data manipulation instructions always push values to the top of the stack and pop results from its top. Our examples feature a register-based virtual machine but the technique we describe should also apply to stack-based virtual machines.

```
void execute_foo(Value stack[])
{
    L0: {
        Instruction instr = { LOADI, 3, 17 };
        int dst = instr.arg1;
        int val = instr.arg2;
        stack[dst] = IntValue(val);
    }
    L1: {
        Instruction instr = { ADD, 0, 1, 2 };
        int dst  = instr.arg1;
        int src1 = instr.arg2;
        int src2 = instr.arg3;
        stack[dst] = DoAdd(stack[src1], stack[src2]);
    }
    L2: {
        Instruction instr = { ADD, 1, 1, 3 };
        int dst  = instr.arg1;
        int src1 = instr.arg2;
        int src2 = instr.arg3;
        stack[dst] = DoAdd(stack[src1], stack[src2]);
    }
    L3: {
        // JUMP
        goto L1;
    }
}
```

Figure 3: Specializing the interpreter to a particular function.

In this paper we are interested in the interpretation overhead that is associated with decoding and dispatching virtual-machine instructions. The decoding overhead comes from fetching the next virtual instruction from memory and computing the values of its parameters; in the example, those would be the `tag` and `arg` fields. The dispatch overhead happens as the interpreter transfers the control to the appropriate instruction handler. The most basic form is a while-switch loop, but some interpreters might use more advanced dispatch techniques such as "threaded code" [6]. The Lua 5.4 interpreter can be configured to use either a portable while-switch loop or a dispatch table using the computed-goto GCC extension.

To minimize the decoding and dispatching overheads, LuaAOT produces a modified version of the inner interpreter loop that is specialized to run a given function. Figure 3 provides an example of this idea. The instructions become compile-time constants and the jumps become goto statements. The `execute_foo` function can be seen as a partial evaluation of the `execute` function, where the `prog` argument is fixed to be the `foo` array from Figure 1. This method to produce a compiler from an interpreter is sometimes called a Futamura Projection [7].

This simple compilation strategy does not optimize all the things that an advanced Lua compiler can try to optimize. For example, there is no attempt to store Lua variables in CPU registers. Similarly to the Lua interpreter, LuaAOT stores all local variables in the Lua stack. However, the simple compilation strategy does provide an

idea of what can be achieved by optimizing the low-hanging fruit. In particular, it can tell us about the interpretation overhead of the original interpreter. Since the bytecode instructions are compile-time constants, the C compiler can use constant propagation to remove most of the operations for decoding the instructions. Similarly, because the Lua jumps are converted to C gotos, we avoid indirect jumps and dispatch tables. The control flow graph also is exposed to the C compiler, possibly allowing further optimizations.

3 THE LuaAOT COMPILER

In this section we describe how we built the LuaAOT compiler and which challenges we encountered in the process. The compiler is free software and the source code is publicly available [9].

The core of the compiler is the `luaot` executable, which takes Lua source code as input and produces the corresponding C code as output. This C code then may be compiled into a Lua extension module, using the same procedure for compiling any Lua extension module written in C. The resulting module may then be loaded by a custom Lua interpreter, which we modified to be able to run compiled functions. It is also possible to produce a standalone executable by bundling the compiled code with a copy of the Lua interpreter. However, compiling entire programs is not typical; we usually compile only the critical parts of the program.

3.1 The Interpreter

The Lua interpreter plays a central role in our system, which comprises of both a compiler and a slightly modified interpreter. There are multiple reasons for this. The first is that programs can contain both compiled and non-compiled sections and the interpreter is necessary to run the non-compiled parts. Moreover, the compiled code also requires the interpreter: because we partially evaluate the inner interpreter loop, the compiled code calls several subroutines from the interpreter. Furthermore, the interpreter code base also houses the Lua runtime and garbage collector, which are used by both the compiled and the non-compiled code.

The custom interpreter has very small changes, compared to the original Lua interpreter. The first modification was to add an additional field to the data structure that represents Lua functions. This field refers to the compiled code for that function, if there is one. The C extension modules that we generate include initialization code that associates the Lua functions with their compiled C implementation.

After this, we told the interpreter how to use these compiled functions. At the start of the `execute` subroutine, just before the inner loop, the interpreter checks whether the function has a compiled version; if so, it transfers the control to that compiled code.

The next change we made is related to the public interface that is exposed to C extension modules. Our partial-evaluation generates code that calls many internal functions from the Lua interpreter, which in normal circumstances are not exposed to extension modules. To allow our generated code to use these internal functions, we modified the interpreter to make all those internal names public.

Finally, we proceeded to implement the code generator, examining the bytecodes one by one. Most had their code directly pasted into the compiler; some required modifications to the generated code, but there was one case that also required modifications to the interpreter: the bytecodes for function calls (CALL, TAILCALL, and RETURN). The Lua 5.4 interpreter has an optimization where Lua-to-Lua calls reuse the same execution frame for the `execute` function. Like a conventional CPU, the Lua interpreter implements these function calls by updating the `prog` argument and the program counter, so that the same interpreter loop naturally runs the called function. Unfortunately, this implementation is incompatible with our compiled code, where each `execute` function is specialized to a particular Lua function. Our solution was to disable this optimization. We believe that with additional work it might have been possible to keep it. However, disabling it was certainly simpler.

We should stress that this change did not harm the tail-recursive functions, which are guaranteed to use O(1) stack space. In the generated C code for the TAILCALL instruction, the crucial function call appears in a tail position, so that a good C compiler can perform the required tail-call optimization.

As important as the changes we made are the many things we did not need to modify. Other than adding a single field to the function objects, we made no other changes to the internal Lua data types. We also did not modify the Lua runtime system, the garbage collector, or the Lua standard library.

3.2 The Code Generator

The code generator receives a Lua module and converts it to C. To do this it calls the Lua bytecode compiler and then it converts the bytecode to C. For each function in the module, the code generator produces an appropriate function header and then it iterates over the function's bytecode, outputting a block of C code for each instruction. For the most part, these blocks of C code are copied verbatim from the original Lua interpreter. However, we had to adapt a few bytecodes. The main categories were bytecodes that modify the program counter, bytecodes using C gotos, and bytecodes related to function calls.

In the original interpreter, jumps are implemented by assigning to the interpreter variable representing the program counter. In our compiler, we replaced each of these jumps by a C goto. This required making the appropriate changes to the generated code for the JUMP instruction, as well as to the instructions that implement for-loops. We also had to change the instructions for binary operations, because of how Lua implements operator overloading. Every binary operation is followed by a special instruction (MMBIN), which handles overloading. When the operands have the expected types (e.g., numbers for the ADD operation), the binary operation just increments the program counter to skip this next instruction. This means that all binary operations contain an implicit jump, which our compiler must also replace by a goto.

The next category of instructions we had to adapt were the ones that use goto statements in their original implementation. One example is the FORCALL instruction, which is always followed by a FORLOOP. As an optimization, the Lua interpreter uses a goto to jump straight to the handler for the FORLOOP, bypassing the usual dispatch logic. Since our compiler is already removing the run-time instruction dispatching, we simply removed this optimization from our generated code.

The instructions for function calls also had the issue of gotos in the original implementation, as we discussed in the previous section.

However, in this case we had to apply the same changes both to the interpreter and to the generated code, so that uncompiled Lua functions could properly call the compiled ones.

3.3 Alternative Compilation Without Gotos

A fragile aspect of LuaAOT is the optimization of replacing Lua jumps by C gotos. While the implementation of the Lua interpreter gave us the opportunity to use gotos without too much trouble, doing this to a different interpreter might have been more difficult. For example, the Lua interpreter is written in C, which is a language with gotos. Had it been written in a different language, it might have been harder to use gotos in the partial evaluation process. Another important aspect is that Lua's core interpreter loop is all in a single function. Had it been broken into smaller subroutines, it might have been more difficult to use gotos because one subroutine would not be able to goto another.

Therefore, we developed an alternative design that implements jumps using a trampoline pattern instead of gotos. As we can see in Figure 4, this trampoline-based implementation also uses a switch-case. However, it dispatches based on the program counter instead of based on the instruction tag. For instructions that don't jump, the handler falls through to the handler for the next instruction. For the instructions that do modify the program counter, the handler ends with a `break` statement, which bounces back to the start of the trampoline.

In terms of implementation effort the trampoline approach is even simpler than the goto-based one. Most of the manual modifications that we described in Section 3.2 involved replacing Lua jumps with C gotos. In the trampoline implementation, the vast majority of those sections can be copy-pasted without any changes at all. The obvious downside of trampolines is that they maintain some of the dispatch-related overhead from the interpreter. We will evaluate this overhead in Section 4.

3.4 Coroutines

Lua's coroutines [4, 5] are a powerful control-flow mechanism, useful for asynchronous programming. They operate in a similar space to features such as generators or delimited continuations. Lua implements coroutines by maintaining a separate call stack for each coroutine. When a coroutine yields, the interpreter saves the current program counter and exits from the interpreter loop (via a longjmp). When the coroutine is resumed, the interpreter must continue the execution loop from where it left.

To make our compiler compatible with coroutines, we had to teach it how to restart the execution from the point where the coroutine was interrupted. To do this, we need to jump to the appropriate location in the code, according to the saved value of the program counter. Figure 5 illustrates how we do this. At the start of the function, we insert a switch-case that jumps to the location indicated by the saved program counter. The rest of the compiled function, including the jump labels, is the same as the version without coroutine support, shown in Figure 3. The switch case is used only once, at the start of the function. After this, all the other jumps happen as previously described, with gotos.

```
void execute_foo(Value stack[])
{
    int pc = 0;
    while (1) {
        switch (pc) {
            case 0: {
                Instruction instr = { LOADI, 3, 17 };
                int dst = instr.arg1;
                int val = instr.arg2;
                stack[dst] = IntValue(val);
                // fallthrough
            }
            case 1: {
                Instruction instr = { ADD, 0, 1, 2 };
                int dst  = instr.arg1;
                int src1 = instr.arg2;
                int src2 = instr.arg3;
                stack[dst] = DoAdd(stack[src1],
                                   stack[src2]);
                // fallthrough
            }
            case 2: {
                Instruction instr = { ADD, 1, 1, 3 };
                int dst  = instr.arg1;
                int src1 = instr.arg2;
                int src2 = instr.arg3;
                stack[dst] = DoAdd(stack[src1],
                                   stack[src2]);
                // fallthrough
            }
            case 3: {
                Instruction instr = { JUMP, 1 };
                pc = instr.arg1;
                break;
            }
        }
    }
}
```

Figure 4: Compilation without gotos, using a trampoline.

```
void execute_foo(Coroutine *f, Value stack[])
{
    switch (f->savedpc) {
        case 0: goto L0;
        case 1: goto L1;
        case 2: goto L2;
        case 3: goto L3;
    }
    L0: { /* ... */ } // LOAD
    L1: { /* ... */ } // ADD
    L2: { /* ... */ } // ADD
    L3: { /* ... */ } // JUMP
}
```

Figure 5: Support for Lua coroutines.

In the alternative trampoline-based compiler, supporting coroutines is even simpler. The trampoline already includes the switch-case that the coroutines need. The only difference is the initial value of the program counter. Instead of always starting from zero (the first instruction), it should start from the value that the coroutine saved.

4 EVALUATION

To evaluate LuaAOT, we studied its performance and we measured the complexity of its implementation. We also made a qualitative analysis of what are the requirements that an interpreter must fulfill to allow a compiler in the style of LuaAOT.

The source code for the benchmarks and the related scripts are available online, in the same repository as the compiler [9].

4.1 Performance

To evaluate the performance of LuaAOT, we compared the running time of the compiled programs with the running time of the interpreter. To provide a baseline, we also compared the results with LuaJIT [16], an advanced just-in-time compiler for Lua.

The benchmarks we used come from the Computer Language Benchmarks Game [8]. We excluded three benchmarks from the list: pydigits, regex-redux, and reverse-complement. The first two require external libraries that are not part of the Lua standard library. The latter is bottlenecked by the string library, which is implemented in C, therefore making it unsuitable for evaluating the performance of the interpreter.

We carried out the measurements on a laptop with an Intel i5-7200U CPU, running Fedora Linux 33. We used Lua version 5.4.3 and LuaJIT version 2.1.0-beta3. For the C compiler we used GCC version 10.3.1, with the -O2 optimization level. For each benchmark we picked an input size large enough to ensure that the fastest implementation took at least one second to run. We ran each benchmark 20 times. The results are summarized in Table 1, which displays the average running time as well as the encountered variation. The error intervals refer to the difference between the average and the maximum or the minimum time, whichever was greater. Figure 6 displays the same running times but normalized by the average time of the Lua interpreter.

In all benchmarks, LuaAOT was faster than the Lua interpreter. The reduction in running time ranged from approximately 20%, in the K-Nucleotide benchmark, to approximately 60%, in the Mandelbrot benchmark. The speed of the trampoline version of LuaAOT fell between the speed of Lua and the speed of the default version of LuaAOT (using gotos). In all cases, LuaJIT was the fastest.

We were curious whether the better performance of LuaAOT compared to Lua was because it ran less CPU instructions or because it could run more instructions per second. To answer this question, we reran the benchmarks using Linux's `perf` tool, which can measure the number of CPU instructions and CPU cycles used by each program. The results are listed in Table 2. They suggest that, at least for this CPU model, the largest factor behind the improved speeds is a reduction in the number of CPU instructions. It appears that the instruction-per-cycle statistic is actually slightly worse for LuaAOT. For most benchmarks the reduction in instruction count is larger than the reduction in time (CPU cycles). We hypothesize that the biggest speedup comes from the compiler removing some of the instructions responsible for bytecode decoding and dispatching. As most of these instructions are cheap (e.g., shifts and masks for decoding), the reduction in the number of instructions is larger than the reduction in cycles, therefore reducing the instructions per cycle.

In theory, removing the bytecode dispatching (and the associated branches) has the potential to improve performance by avoiding costly branch mispredictions. However, in our benchmarks this was not a big factor, because the CPU already did a good job of predicting the branches in the interpreted version. In almost all benchmarks, the branch miss rates for both Lua and LuaAOT were under 1%; the sole exception is the the N-Body benchmark for Lua, with a branch miss rate of 1.2%. With these low rates of branch miss, we don't think it is meaningful to compare the absolute numbers. It is also hard to tell whether the compilation is helping the branch-miss rates.

One trade-off of LuaAOT compared to the interpreter is that the generated binaries are larger than the corresponding bytecode. To evaluate this aspect of the compiler we measured the sizes of the bytecode-compiled and AOT-compiled version of each benchmark. The results are listed in Table 3. The sizes of the compiled code were much larger than the sizes of the bytecodes, albeit not so large that it becomes prohibitive. Nevertheless, programmers may want to consider not compiling the parts of the program that are not performance-sensitive.

4.2 Complexity of the implementation

One of the selling points of the partial-evaluation strategy that we used is its extreme simplicity. To measure this, we counted the lines of code of our code generator, as a proxy for implementation complexity.

We built the code generator by hand, using generous amounts of copy-pasting of code from the Lua interpreter loop and from the Lua bytecode compiler. We believe that, if desired, it should be possible to automate a large portion of this work. That would expedite the process of updating LuaAOT to new versions of the Lua interpreter.

Because we copied some subroutines from the Lua bytecode compiler, we chose to write the code generator in C. Out of the total of 1600 lines of code in the generator, 450 lines can be attributed to those subroutines from the bytecode compiler, which are responsible for traversing and printing bytecodes. Code templates derived from the core interpreter loop account for over half of the code generator, about 850 lines. The rest of the code, which we wrote from scratch, fits in less than 500 lines. It consists of miscellaneous things such as comments, command-line option handling, and the initialization routines for the generated extension modules.

For comparison, the reference Lua interpreter contains 28 thousand lines of C code [12] and the LuaJIT just-in-time compiler has 80 thousand lines of C and 35 thousand lines of platform-specific assembly language [16]. Another thing that we can point out is that while LuaAOT required us to be familiar with the internals of the Lua interpreter, the final product did not require complex analysis algorithms or optimization passes.

Benchmark	Lua	Trampoline	LuaAOT	LuaJIT
Binary Trees	3.51 ± 0.03	2.77 ± 0.37	2.68 ± 0.10	1.46 ± 0.15
Fannkuch	44.31 ± 0.13	33.45 ± 0.17	21.92 ± 0.17	7.10 ± 0.02
Fasta	5.05 ± 0.08	4.21 ± 0.28	3.84 ± 0.36	1.08 ± 0.01
K-Nucleotide	4.14 ± 0.14	3.70 ± 0.10	3.44 ± 0.12	1.18 ± 0.09
Mandelbrot	16.27 ± 0.08	16.41 ± 0.22	6.17 ± 0.31	1.78 ± 0.00
N-Body	18.23 ± 1.85	16.13 ± 0.86	12.14 ± 0.49	1.13 ± 0.02
Spectral Norm	39.86 ± 0.27	27.02 ± 0.12	19.00 ± 0.78	1.27 ± 0.01

Table 1: Running times, in seconds.

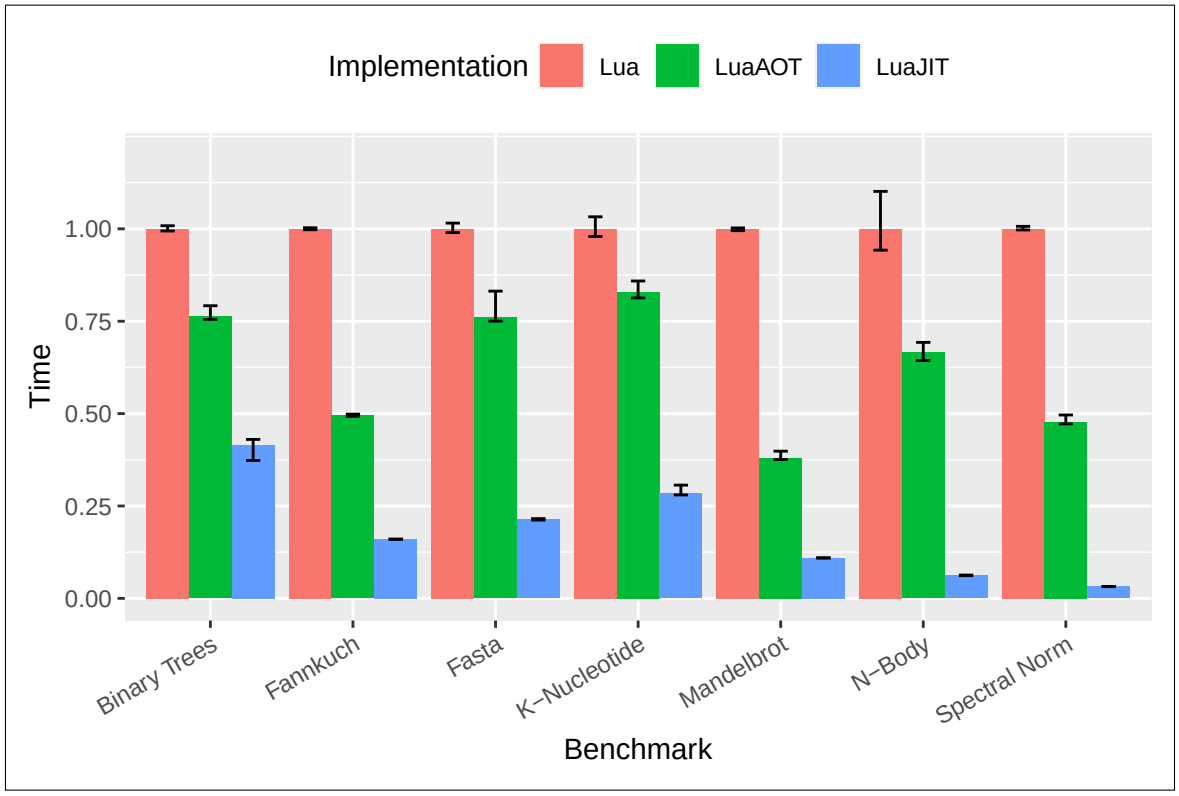

Figure 6: Normalized running times.

Benchmark	Instrs (%)	Time (%)
Binary Trees	77.9	76.4
Fannkuch	44.0	49.6
Fasta	69.4	78.8
K-Nucleotide	75.6	83.7
Mandelbrot	32.5	37.9
N-Body	59.7	64.5
Spectral Norm	47.4	47.4

Table 2: CPU instruction count for LuaAOT, relative to Lua.

4.3 Applicability of the technique

While the work we have presented is specific to the reference Lua interpreter, we think that the technique is simple enough to be applicable to other dynamic language interpreters. In this section, we discuss what were the aspects of Lua that our interpreter relied on, and what conditions are necessary to apply this to another interpreter. Of course, the performance improvements will depend on the specifics of the interpreter, in particular what percentage of time is attributable to the core interpreter loop.

The first important thing is that our technique would not work as easily for AST-based interpreters. While it is also possible to

Benchmark	Bytecode (KB)	AOT (KB)
Empty File	0.07	160
Mandelbrot	0.9	282
Fannkuch	1.1	302
Binary Trees	1.4	298
Spectral Norm	1.5	317
K-Nucleotide	2.2	368
Fasta	3.1	533
N-Body	3.2	619

Table 3: Size of compiled modules.

use partial evaluation for an AST-based interpreter, it is more complicated than for a bytecode-based one because of the frequent presence of recursion in the main interpreter loop.

Since our technique is based on partial evaluation, the language used to implement the original interpreter is important. C, which is a popular language for writing interpreters, worked well for several reasons: the presence of a goto statement, the availability of optimizing compilers, and the existence of preprocessor macros.

When we compile jump instructions in the bytecode, we want a similar jump operation in our target language. In C we can use goto statements for this purpose, provided that the control flow in the original interpreter is all inside a single interpreter function. If the target language does not have goto statements, it is harder to compile the unstructured jumps in the bytecode.

Using C as the target language allowed us to take advantage of several optimizations from the C compiler, including constant propagation for the bytecode instructions. This gives to the partial evaluator the luxury of emitting code that is almost identical to the code used by the original interpreter. This would be harder to do if, for example, the original interpreter were implemented in hand-written assembly language. In that case we would likely have to implement the constant propagation ourselves.

Albeit not a fundamental requirement, the C preprocessor was a convenient feature. The LuaAOT code generator is essentially a text-based code transformer and in that context it helps to have a text-based macro system built into the target language. Unlike inline functions, macros can jump to other parts of the program and assign directly to local variables.

5 RELATED WORK

Although our work is inspired by partial evaluation, it is not a partial evaluation system. There is a rich literature on these partial evaluation systems and their application to interpreters [1, 13]. However, one difference between our work and these partial evaluation systems is that they usually require that the input interpreter must be in some specific format that the partial evaluator can work with. LuaAOT is a case study in doing this partial evaluation in an ad-hoc manner, on an existing interpreter.

A relevant example of partial evaluation for interpreters is the Truffle framework [18, 19]. Truffle allows a language implementer to create an efficient just-in-time compiler based on an AST interpreter. The implementer can write the AST interpreter and provide hints that tell Truffle which run-time type information should be collected and where to use the partial evaluation. As we just mentioned, one important difference compared to our work is that Truffle requires that the interpreter be written in Java, using the Truffle framework.

Another area we touch is the study of interpretation overhead. One way that this has been studied is by profiling the interpreter while it is running, to measure how much of the execution time can be assigned to bytecode decoding and dispatching [14]. However, if the motivation for the question is to compare the performance of an interpreter with a compiler, it is useful to measure the result of a compiler, which in addition to removing the decoding and dispatching, would also perform optimizations that are natural to implement in a compiler. One such compiler is Barany's pylibjit [2]. Barany implemented a just-in-time compiler for Python using the GNU LibJIT [17] library. Similarly to LuaAOT, his compiler also works at the bytecode level, converting each Python bytecode instruction into a machine code sequence. Barany measured the effect of enabling and disabling various optimizations passes of his compiler, to estimate how much of an impact these aspects have in the performance of Python programs [3]. Some of the optimizations that he implemented were removal of redundant reference counting, static dispatch of arithmetic operations, unboxing of number and container objects, and call-stack frame removal. These Python-specific optimizations allowed Barany to investigate the performance impact of more features of the interpreter other than just the bytecode handling. However, his compiler is more complex than ours; he had to reimplement most of the bytecodes using the LibJIT framework.

Finally, we want to mention that our work does not consider optimizations of the interpreter itself, including superinstructions or threaded dispatch [6]. We also did not study the effect of type inference. Our focus was in reducing the direct interpretation overhead.

6 CONCLUSION

Dynamic programming languages are often implemented using interpreters, which spend some portion of the running time on interpretation overhead. Compilers can avoid this, but they may be complex to implement.

In this paper we have presented LuaAOT, a simple ahead-of-time compiler for Lua. Using less than 500 lines of new code in a total of 1600 lines, we were able to compile the entirety of Lua, including features like coroutines and tail calls.

In return, we achieved a reduction in running times between 20% and 60%. While these numbers cannot compete head-to-head with a good JIT compiler, they demonstrate a noticeable performance boost, for a tiny fraction of the implementation cost. These numbers also offer a contribution to studies about interpretation overheads.

One novelty presented in this paper was the trampoline-based compilation strategy, which does not require modifications to the jump instructions. While it is not as fast as the goto-based strategy, it is even simpler to implement. It also supports coroutines with very little work. That is something that is tricky to do when compiling to C, because C itself does not support coroutines.

We believe that the technique we used to implement LuaAOT may be of interest to other programming languages. In particular, the basic ideas of our work may also be applicable to other interpreters that use a bytecode-based virtual machine written in C.

ACKNOWLEDGMENTS

This work has been funded in part by CNPQ, grants 153918/2015-2 and 305001/2017-5; and by CAPES, grant number 001.

A EXAMPLE OF GENERATED CODE

In Section 2, our examples featured a simplified interpreter. In this appendix, we show some real code produced by LuaAOT. It is the result of compiling the `foo` function from Figure 1. To make it fit, we made minor stylistic edits and replaced some sections by `/*...*/` comments. The code starts with some boilerplate initialization code, followed by the coroutine dispatch table, and finally the handlers for each bytecode instruction. In this code, we can see some of the preprocessor macro tricks. The `vmfetch` macro initializes the `i` variable to a compile-time constant, allowing the C compiler to constant fold the `GETARG_sBx` macro. The `LUAOT_SKIP1` macro allows instructions like `op_arith` to skip over the next instruction; in the original interpreter, they increment the program counter (`pc++`) while in LuaAOT we replace that by `goto LUAOT_SKIP1`.

```
static void execute_foo(lua_State *L, CallInfo *ci)
{
  int trap = L->hookmask;
  LClosure *cl = clLvalue(s2v(ci->func));
  TValue *k = cl->p->k;
  const Instruction *pc = ci->u.l.savedpc;
  if (trap) { /*...*/ }
  StkId base = ci->func + 1;

  Instruction *code = cl->p->code;
  Instruction i;
  StkId ra;

  switch (pc - code) {
    case 0: goto label_00;
    /*...*/
    case 6: goto label_06;
  }

  // 0 - LOADI 3 17
  #undef  LUAOT_SKIP1
  #define LUAOT_SKIP1 label_02
  label_00: {
    aot_vmfetch(0x80080181);
    lua_Integer b = GETARG_sBx(i);
    setivalue(s2v(ra), b);
  }

  // 1 - ADD 0 1 2
  #undef  LUAOT_SKIP1
  #define LUAOT_SKIP1 label_03
  label_01: {
    aot_vmfetch(0x02010022);
    op_arith(L, l_addi, luai_numadd);
  }

  /*...*/
  // 6 - RETURN0
  label_06: {
    aot_vmfetch(0x00010247);
    if (L->hookmask) {
      L->top = ra;
      savepc(ci);
      luaD_poscall(L, ci, 0);
      trap = 1;
    } else {
      L->ci = ci->previous;
      L->top = base - 1;
      for (int nres = ci->nresults; nres > 0; nres--)
        setnilvalue(s2v(L->top++));
    }
    return;
}
```

Figure 7: The actual code generated by the compiler.

REFERENCES

[1] Lars Ole Andersen. 1992. Partial evaluation of C and automatic compiler generation. In *Compiler Construction*, Uwe Kastens and Peter Pfahler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 251–257.

[2] Gergö Barany. 2014. pylibjit: A JIT Compiler Library for Python.. In *Software Engineering (Workshops)*. 213–224.

[3] Gergö Barany. 2014. Python Interpreter Performance Deconstructed. In *Proceedings of the Workshop on Dynamic Languages and Applications (Dyla'14)*. 5:1–5:9. https://doi.org/10.1145/2617548.2617552

[4] Ana Lúcia de Moura. 2004. *Revisitando co-rotinas.* Ph.D. Dissertation. PUC-Rio.

[5] Ana Lúcia de Moura, Noemi Rodriguez, and Roberto Ierusalimschy. 2004. Coroutines in Lua. *Journal Universal Computer Science* 10, 7 (July 2004), 910–925. https://doi.org/10.3217/jucs-010-07-0910

[6] Anton M. Ertl and David Gregg. 2003. The Structure and Performance of Efficient Interpreters. *Journal of Instruction-Level Parallelism* 5 (Nov. 2003). https://www.jilp.org/vol5/index.html

[7] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391. https://doi.org/10.1023/A:1010095604496

[8] Isaac Gouy. 2013. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame

[9] Hugo Gualandi. 2021. LuaAOT 5.4 source code repository. https://github.com/hugomg/lua-aot-5.4

[10] Hugo Gualandi and Roberto Ierusalimschy. 2020. Pallene: A companion language for Lua. *Science of Computer Programming* 189 (2020), 102393. https://doi.org/10.1016/j.scico.2020.102393

[11] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2005. The Implementation of Lua 5.0. *Journal Universal Computer Science* 11, 7 (July 2005), 1159–1176. https://doi.org/10.3217/jucs-011-07-1159

[12] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2021. Lua 5.4 interpreter source code. https://www.lua.org/versions.html

[13] Neil D. Jones. 2004. Transformation by interpreter specialisation. *Science of Computer Programming* 52, 1 (2004), 307–339. https://doi.org/10.1016/j.scico.2004.03.010 Special Issue on Program Transformation.

[14] Nagy Mostafa, Chandra Krintz, Calin Cascaval, David Edelson, Priya Nagpurkar, and Peng Wu. 2010. *Understanding the Potential of Interpreter-based Optimizations for Python.* Technical Report. University of California, Santa Barbara. https://www.cs.ucsb.edu/sites/cs.ucsb.edu/files/docs/reports/2010-14.pdf

[15] Peter Sestoft Neil D. Jones, Carsten K. Gomard. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice Hall.

[16] Mike Pall. 2005. LuaJIT, a Just-In-Time Compiler for Lua. http://luajit.org/luajit.html http://luajit.org/luajit.html.

[17] Rhys Weatherley. 2004. GNU LibJIT. The GNU LibJIT library.

[18] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, 662–676. https://doi.org/10.1145/3062341.3062381

[19] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. 187–204. https://doi.org/10.1145/2509578.2509581